Chapter 15: Object-Oriented Programming (OOP)

15.6 Inheritance and The Rule of 3 (5)

copy constructor, copy assignment operator and destructor (move constructor, move assignment operator)

How does inheritance interact with dynamic memory allocation (the use of `new` and `delete`)? For example, if a base-class uses dynamic memory allocation and redefines the copy constructor and the copy assignment operator, how does that affect the implementation of the derived-class?

The answer depends on the nature of the derived-class.

- If the derived-class does not allocate dynamic memory, you need not take any action.
- If the derived-class does allocate dynamic memory, **additional actions required!**

Let’s look at the latter case. Consider a base-class `DoubleArray` that manages a dynamic `double` array and a derived-class `DoubleArrayID` that inherits the base-class and adds a random ID to each element (the `DoubleArrayID` has a dynamic `int` array).

The base-class `DoubleArray` follows what we have learned before, and we utilize the `nullptr` to help us check the status of the pointer for the `double` array.

The `nullptr` is a keyword that can be used at all places where `NULL` is expected[1]. Like `NULL`, `nullptr` is implicitly convertible and comparable to any pointer type. **Unlike `NULL`, it is not implicitly convertible or comparable to integral types**.

DoubleArray.h

```
#ifndef DOUBLE_ARRAY_H
#define DOUBLE_ARRAY_H
#include <iostream>

class DoubleArray {
public:
    DoubleArray() = default;
    DoubleArray(unsigned in_size, double val = 0.0);
    DoubleArray(const DoubleArray& rhs); //copy constructor
    // copy assignment operator
    DoubleArray& operator=(const DoubleArray& rhs);
    virtual void print(std::ostream&) const;
    virtual ~DoubleArray();

protected:
    double* raw_data = nullptr;
```

[1] Details on `nullptr` https://www.geeksforgeeks.org/understanding-nullptr-c/

```
    unsigned array_size = 0;
};

std::ostream& operator << (std::ostream&, const DoubleArray&);

#endif
```

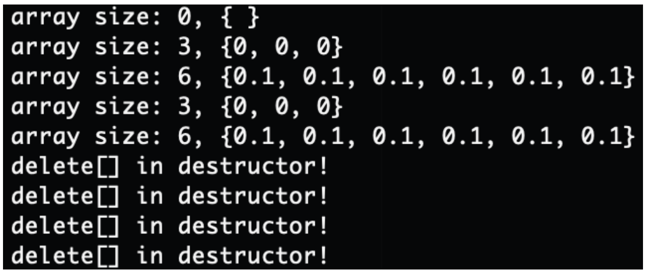
DoubleArray.cpp

```
#include "DoubleArray.h"
#include <iostream>
using namespace std;

DoubleArray::DoubleArray(unsigned in_size, double val) :
array_size(in_size) {
    raw_data = new double[array_size];
    for (unsigned index = 0; index != array_size; ++index)
        raw_data[index] = val;
}

DoubleArray::DoubleArray(const DoubleArray& rhs) {
    array_size = rhs.array_size;
    raw_data = new double[array_size];
    for (unsigned index = 0; index != array_size; ++index)
        raw_data[index] = rhs.raw_data[index];
}

DoubleArray& DoubleArray::operator = (const DoubleArray& rhs) {
    if (this == &rhs)
        return *this;
    if (raw_data != nullptr)
        delete[] raw_data;
    array_size = rhs.array_size;
    raw_data = new double[array_size];
    for (unsigned index = 0; index != array_size; ++index)
        raw_data[index] = rhs.raw_data[index];
    return *this;
}

void DoubleArray::print(ostream& os) const {
    os << "array size: " << array_size << ", {";
    if(array_size > 0){
      for (unsigned index = 0; index < array_size - 1; ++index){
         os << raw_data[index] << ", ";
      }
      os << raw_data[array_size - 1] << "}" << endl;
    }
    else os << " }" <<  endl;
}

DoubleArray::~DoubleArray() {
    if (raw_data != nullptr){
        delete[] raw_data;
        std::cout << "delete[] in destructor!" << std::endl;
    }
}

ostream& operator << (ostream& os, const DoubleArray& a) {
    a.print(os);
    return os;
}
```

We can test the base-class through the following client code:

TestDoubleArray.cpp

```
#include "DoubleArray.h"
#include <iostream>
using namespace std;

int main() {
    DoubleArray da0;
    DoubleArray da1(3);
    DoubleArray da2(6, 0.1);
    DoubleArray da3 = da1;
    DoubleArray da4;
    da4 = da2;
    cout << da0 << da1 << da2 << da3 << da4;
    return 0;
}
```



```
array size: 0, { }
array size: 3, {0, 0, 0}
array size: 6, {0.1, 0.1, 0.1, 0.1, 0.1, 0.1}
array size: 3, {0, 0, 0}
array size: 6, {0.1, 0.1, 0.1, 0.1, 0.1, 0.1}
delete[] in destructor!
delete[] in destructor!
delete[] in destructor!
delete[] in destructor!
```

Remark: in the implementation of the assignment operator and destructor, we utilize the nature of `nullprt` to help us check the status of `raw_data` pointer and to take further actions if it is not a `nullptr`.

Derived-class Using `new` for Member

Copy Constructor and the Copy Assignment Operator

For the copy constructor and the copy assignment operators of the derived-class, the following are what to pay attention to:

- Make copy assignment non-virtual, take the parameter by const&, and return by non-const&
- A copy operation should copy
- Make copy assignment safe for self-assignment

If the derived-class `DoubleArrayID` DOES also allocate dynamic memory, then **there are a couple of new tricks to learn**.

DoubleArrayID.h

```
#ifndef DOUBLE_ARRAYID_H
#define DOUBLE_ARRAYID_H
#include "DoubleArray.h"

class DoubleArrayID : public DoubleArray{
public:
    DoubleArrayID() = default;
    DoubleArrayID(unsigned in_size, double val = 0.0);
    DoubleArrayID(const DoubleArrayID& rhs);
    DoubleArrayID& operator=(const DoubleArrayID& rhs);
    void print(std::ostream&) const;
    ~DoubleArrayID();

protected:
    int* randID = nullptr;
};

#endif
```

Derived-class Destructor

When a derived-class destructor finishes, it automatically calls the base-class destructor[2].

```
DoubleArrayID::~DoubleArrayID(){
    if (randID != nullptr)
        delete[] randID;
}
```

Derived-class Constructor

```
DoubleArrayID::DoubleArrayID(unsigned in_size, double val) :
                                            DoubleArray(in_size, val){
    randID = new int[array_size];
    srand(0); // set random seed
    for (unsigned index = 0; index != array_size; ++index)
        randID[index] = rand()%100;
}
```

Derived-class Copy Constructor

```
DoubleArrayID::DoubleArrayID(const DoubleArrayID& rhs) :
                                                  DoubleArray(rhs){
    randID = new int[array_size];
    for (unsigned index = 0; index != array_size; ++index)
        randID[index] = rhs.randID[index];
}
```

The `DoubleArrayID` copy constructor invokes the `DoubleArray` copy constructor to handle the `DoubleArray` part of the data.

Q1: What happens if you don't invoke the `DoubleArray` copy constructor?
A1: It will call the `DoubleArray` default constructor. (shallow copy)

[2] https://www.geeksforgeeks.org/order-constructor-destructor-call-c/

Q2: Note that the member initializer list passes a `DoubleArrayID` reference to a `DoubleArray` copy constructor. Is it ok?
A2: Yes, since upcasting with reference or pointer is always ok. In other words, `DoubleArrayID` is-a `DoubleArray`.

Derived-class's Copy Assignment Operator

```
DoubleArrayID& DoubleArrayID::operator = (const DoubleArrayID&
rhs){
    if (this == &rhs)
        return *this;
    this->DoubleArray::operator=(rhs); // copy base portion
    if (randID != nullptr)
        delete[] randID;
    randID = new int[array_size];
    for (unsigned index = 0; index != array_size; ++index)
        randID[index] = rhs.randID[index];
    return *this;
}
```

Q: Instead of `DoubleArray::operator=(rhs)`, can you do `*this = rhs`?
A: No, the compiler will use `DoubleArrayID::operator=(rhs)` and create a recursive call (infinite loop!).

Let us now put together the `DoubleArrayID` implementation:

DoubleArrayID.cpp

```
#include "DoubleArrayID.h"
#include <iostream>
#include <ctime> // random seed
#include <cstdlib> //random pick
using namespace std;

DoubleArrayID::DoubleArrayID(unsigned in_size, double val) :
DoubleArray(in_size, val){
    randID = new int[array_size];
    srand(time(0)); // random seed
    for (unsigned index = 0; index != array_size; ++index)
```

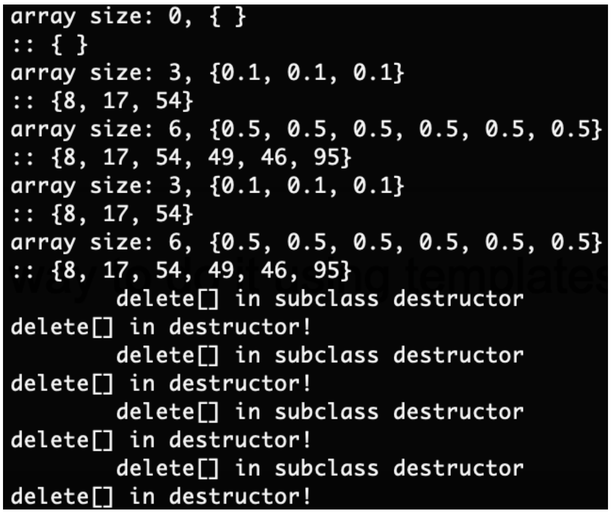
```
        randID[index] = rand()%100;
}

DoubleArrayID::DoubleArrayID(const DoubleArrayID& rhs) :
                                                       DoubleArray(rhs){
    randID = new int[array_size];
    for (unsigned index = 0; index != array_size; ++index)
        randID[index] = rhs.randID[index];
}

DoubleArrayID& DoubleArrayID::operator = (const DoubleArrayID&
rhs){
    if (this == &rhs)
        return *this;
    DoubleArray::operator=(rhs); // copy base portion
    if (randID != nullptr)
        delete[] randID;
    randID = new int[array_size];
    for (unsigned index = 0; index != array_size; ++index)
        randID[index] = rhs.randID[index];
    return *this;
}

void DoubleArrayID::print(ostream& os) const {
    DoubleArray::print(os);
    os << ":: {";
    if(array_size > 0){
        for (unsigned index = 0; index < array_size - 1; ++index){
            os << randID[index] << ", ";
        }
        os << randID[array_size - 1] << "}" << endl;
    } else os << " }" << endl;
}

DoubleArrayID::~DoubleArrayID(){
    if (randID != nullptr){
        delete[] randID;
        cout << "\tdelete[] in subclass destructor" << endl;
    }
```

```
}
```

And we can test the derived-class through the following client code:

TestDoubleArrayID.cpp

```
#include "DoubleArray.h"
#include "DoubleArrayID.h"
#include <iostream>
using namespace std;

int main(){
    DoubleArrayID dar0;
    DoubleArrayID dar1(3, 0.1);
    DoubleArrayID dar2(6, 0.5);
    DoubleArray* dar3 = new DoubleArrayID(dar1);
    DoubleArrayID dar4;
    dar4 = dar2;

    cout << dar0 << dar1 << dar2 << *dar3 << dar4;

    delete dar3;
    return 0;
}
```



```
array size: 0, { }
:: { }
array size: 3, {0.1, 0.1, 0.1}
:: {8, 17, 54}
array size: 6, {0.5, 0.5, 0.5, 0.5, 0.5, 0.5}
:: {8, 17, 54, 49, 46, 95}
array size: 3, {0.1, 0.1, 0.1}
:: {8, 17, 54}
array size: 6, {0.5, 0.5, 0.5, 0.5, 0.5, 0.5}
:: {8, 17, 54, 49, 46, 95}
        delete[] in subclass destructor
delete[] in destructor!
        delete[] in subclass destructor
delete[] in destructor!
        delete[] in subclass destructor
delete[] in destructor!
        delete[] in subclass destructor
delete[] in destructor!
```

Move Constructor and the Move Assignment Operator

For the move constructor and the move assignment operators of the derived-class, the following are what to pay attention to[3]:

[3] Guidelines for copy and move https://www.modernescpp.com/index.php/c-core-guidelines-copy-and-move-

- Make move assignment non-virtual, take the parameter by &&, and return by non-const&
- A move operation should move and leave its source in a valid state
- Make move assignment safe for self-assignment

The std::move() function

The `std::move` function should be used when implementing the move constructor to move the information in base classes or composed objects to their new destination object[4]. The `static_cast` is also utilized to convert derived class to based class[5].

DoubleArray.h

```
class DoubleArray {
public:
    // …
    DoubleArray(DoubleArray && source); // move constructor
    DoubleArray& operator=(DoubleArray && source);
    // …
};
```

DoubleArray.cpp

```
// Move Constructor
DoubleArray::DoubleArray(DoubleArray && source)
    : raw_data( source.raw_data ){
    cout << "Base-Class Move Constructor" << endl;
    source.raw_data = nullptr;
}

// Move Assignment Operator
DoubleArray& DoubleArray::operator=(DoubleArray && source){
    if(this != &source){
        delete[] this->raw_data;
        this->raw_data = source.raw_data;
        source.raw_data = nullptr;
    }
    cout << "Base-class Move Assignment" << endl;
```

rules

[4] Move constructor and move assignment operator: http://www.icce.rug.nl/documents/cplusplus/cplusplus13.html

[5] Static_cast https://www.geeksforgeeks.org/static_cast-in-c-type-casting-operators/#:~:text=Static%20Cast%3A%20This%20is%20the,For%20e.g.

```
    return *this;
}
```

DoubleArrayID.h

```
class DoubleArrayID : public DoubleArray{
public:
    // …
    DoubleArrayID(DoubleArrayID && source);
    DoubleArrayID& operator=(DoubleArrayID && source);
    // …
};
```

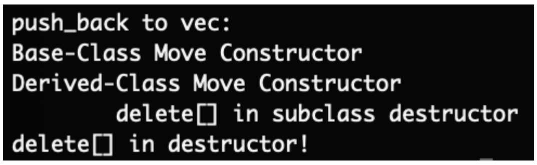
DoubleArrayID.cpp

```
DoubleArrayID::DoubleArrayID(DoubleArrayID && source)
    : DoubleArray(std::move(source)), randID(source.randID){
    cout << "Derived-Class Move Constructor" << endl;
    source.randID = nullptr;
}

// Move Assignment Operator
DoubleArrayID& DoubleArrayID::operator=(DoubleArrayID && source){
    if(this != &source){
        // use base-class' move assignment
        static_cast<DoubleArray &>(*this) = std::move(source);
        // (DoubleArray &)(*this) = std::move(source); // c-style
        delete[] this->randID;
        this->randID = source.randID;
        source.randID = nullptr;
    }
    cout << "Derived-class Move Assignment" << endl;
    return *this;
}
```

TestDoubleArrayIDMoveCon.cpp

```
#include "DoubleArray.h"
#include "DoubleArrayID.h"
#include <iostream>
#include <vector>
using namespace std;
```

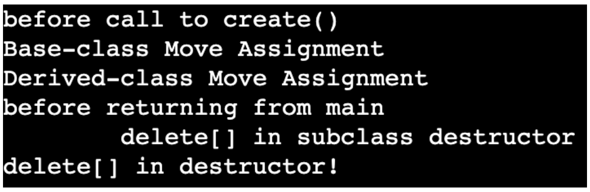
```
int main(){
    vector<DoubleArrayID> vec;
    vec.reserve(2); // reserve space for 2

    // Inserting Object of Move Class
    cout << "push_back to vec:" << endl;
    vec.push_back(DoubleArrayID(3, 0.1));
    return 0;
}
```



```
push_back to vec:
Base-Class Move Constructor
Derived-Class Move Constructor
        delete[] in subclass destructor
delete[] in destructor!
```

<u>TestDoubleArrayIDMoveAss.cpp</u>

```
#include "DoubleArray.h"
#include "DoubleArrayID.h"
#include <iostream>
using namespace std;

DoubleArrayID create(){
    return DoubleArrayID(3, 0.5);
}

int main(){
    DoubleArrayID m;
    cout << "before call to create()" << endl;
    m = create();
    cout << "before returning from main" << endl;
    return 0;
}
```



```
before call to create()
Base-class Move Assignment
Derived-class Move Assignment
before returning from main
        delete[] in subclass destructor
delete[] in destructor!
```

Summary

When both the base-class and the derived-class allocate dynamic memory, the derived-class destructor, copy/move constructor, and copy/move assignment operator all must use their base-class counterparts to handle the base-class component.

- For a virtual destructor, it is done automatically.
- For a copy constructor, it is accomplished by invoking the base-class' copy constructor in the member initialization list, or else the default constructor is invoked automatically.
- For the copy assignment operator, it is accomplished by using the scope-resolution operator in an explicit call of the base-class' copy assignment operator.
- For the move constructor and move assignment operator, they are similar to the copy. However, there is the `std::move` function and `static_cast` that is required.

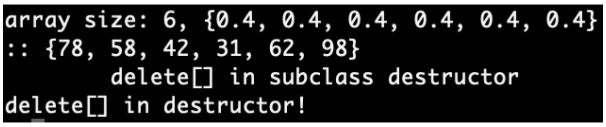
The Need for Virtual Destructor for Base-class

Consider the following client code with some print-out message when we invoke the destructors:

UseVirtualDestructor.cpp

```
#include "DoubleArray.h"
#include "DoubleArrayID.h"
#include <iostream>
using namespace std;

int main(){
    DoubleArray* dar = new DoubleArrayID(6, 0.4);
    cout << *dar;
    delete dar;
    return 0;
}
```

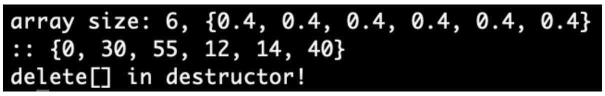
Output:



```
array size: 6, {0.4, 0.4, 0.4, 0.4, 0.4, 0.4}
:: {78, 58, 42, 31, 62, 98}
        delete[] in subclass destructor
delete[] in destructor!
```

The code uses `delete` to free the objects allocated by `new` illustrates why the base-class should have a `virtual` destructor.

- If the destructors are not `virtual`, then just the destructor corresponding to the pointer type is called. This means that only the `DoubleArray` destructor would be called, even if the pointer pointed to a `DoubleArrayID` object.



```
array size: 6, {0.4, 0.4, 0.4, 0.4, 0.4, 0.4}
:: {0, 30, 55, 12, 14, 40}
delete[] in destructor!
```

- If the destructors are `virtual`, the destructor corresponding to the object type is called[6]. So, if a pointer points to a `DoubleArrayID` object, the `DoubleArrayID` destructor is called. When a `DoubleArrayID` destructor finishes, it automatically calls the base-class destructor[7].
- Thus, `virtual` destructors ensure that the correct sequence of destructor calls is followed.

[6] You need to make base-class destructor virtual. https://www.geeksforgeeks.org/virtual-destructor/

[7] Order of destructor chaining in C++ https://www.geeksforgeeks.org/order-constructor-destructor-call-c/